Ryan Hubbuck

CS-320

10/09/2023

Project 2

Having recently finished the initial version of our mobile application, we should now take the opportunity to reflect upon the work our organization recently completed. The application consists of three key components: a contact, task, and appointment service. Each service is composed of a related class and service class, and each is also associated with a testing class.

First, we had to ensure that Task, Contact, and Appointment objects and associated service classes were aligned with the overall software requirements. Each object was required to contain a unique ID to ensure functionality of adding, searching for, and deleting individual objects, so JUnit tests were written to check for null IDs and unique IDs. Additionally, each attribute of each object was tested to ensure that strings were of the appropriate length, and further testing was added to ensure that Appointment dates could not be added if they were to be in the past. An example of using assertTrue is below to ensure that an ID is unique for the Task object:

@Test

void testTaskClassIdNull() {

Assertions.assertThrows(IllegalArgumentException.class, () -> {

new Task(null, "Ryan Hubbuck", "Just a description");

});

}

Additionally, the processes of creating, updating, and deleting individual Tasks and Contacts were verified through JUnit to only do so by finding the specific id value. The Appointment class did not allow for object updating, but object creation and deletion was handled in the same fashion. These tests ensure that the software will work per outlines to only create, update, or delete correct objects with a matching id.

The way in which we were able to verify the effectiveness of our JUnit tests was by inspecting the pass/fail case for each test and the overall test coverage. Within each test file we were able to inspect and verify that each test case ran correctly and responded with the correct assertion. An example can be seen within TaskServiceTest, which runs and completes 5/5 tests. The goal is to get as close to 100% test coverage as possible, and nearly every one of our test files runs with over 90% coverage. Achieving over 90% coverage shows that our implemented tests are verifying nearly all code written within the class and service files. Writing effective JUnit tests allow us to ensure our program is aligned with the software requirements such as

new objects only being added if they contain a unique id, objects being deleted by id, and object attributes being properly updated by id.

To ensure our code was technically sound we followed all specifications outlined per the instructions and ran individual modules to ensure objects were created, updated, and deleted as expected. Each module was written incrementally and utilized in-line comments to ensure all team members were on the same page and high-quality work was being completed. Code was compiled and executed following the completion of each individual method. This ensured that the code being written was technically sound and efficient. An example of an object deletion test can be seen below:

// Test deletion of task by finding matching id

@Test

void testTaskServiceDelete() {

TaskService.addTask("01", "Task name", "Task One");

// delete task by id

TaskService.deleteTask("01");

// loop through the task list if there are any objects

for (int i = 0; i < TaskService.taskList.size(); i++) {

// make sure the object no longer exists with specified id

assertFalse(TaskService.taskList.get(i).getId().equals("01"));

}

}

This code block demonstrates the technical soundness of our program. In-line comments explain each step in which a loop is utilized to examine the ID of each Task object in the data structure to verify that the object with the specified ID was deleted. We were able to ensure that all tests were executed correctly, and all technical program requirements were satisfied.

Our code is also efficient in addition to being technically sound. While the prompts were open-ended and does not explicitly outline which data structure to use, we chose to utilize an ArrayList due to its efficiency in adding and deleting single objects, as well as its efficiency in looping through all objects. After creating getter and setter methods within the respective Task and Contact classes (except for id, which is not updatable), we then used these methods to get the attributes of each object effectively and efficiently to check their respective id’s and set new attribute values. An example is below:

// test updating Task description by finding matching id

@Test

void testContactServiceUpdateDescription() {

TaskService.addTask("007", "Testing Name", "Old desciption");

TaskService.updateDescription("007", "New description");

// Loop through task list

for (int i = 0; i < TaskService.taskList.size(); i++) {

// look for matching id of the object added for this test

if (TaskService.taskList.get(i).getId().equals("007")) {

// Verify that the specified Task description has been changed

assertTrue(TaskService.taskList.get(i).getDescription().equals("New description"));

}

}

}

As seen above, the data structure of an ArrayList allows effective time and space management in finding objects with a specific id, and combining this with methods created such as getDescription() allows for efficient test-writing and execution. Although Appointment objects were not able to be updated per program guidelines, we were required to test the date attributes for validity. In order to complete this most efficiently, the built in java.util.Date class was imported and implemented. An example of testing for a valid date can be seen below:

// test for date in the past

@Test

void testAppointmentClassDateInPast() {

Assertions.assertThrows(IllegalArgumentException.class, () -> {

// will provide date 24 hours in the past

new Appointment("001", new Date(System.currentTimeMillis() - 24 \* 60 \* 60 \* 1000L), "Just a description");

});

}

Using this built-in utility was the most effective and efficient manner to complete the task and ensure proper program behavior.

The JUnit tests written within our program were white-box tests because an understanding of the internal structure and inner-working of the code was required to successfully create test cases and measure outcomes. White-box testing involves testing the internal logic of the program, and examples of this would be statement and branch coverage. As seen within the AppointmentServiceTest, for example, this testing was used to ensure performance and security by discovering errors within software design, logic, and implementation. A few of the specific JUnit test methods implemented in these modules were assertTrue, assertFalse, and assertThrows. assertTrue is used to verify that a condition results to true at runtime, while assertFalse does the opposite and verifies that a condition results to false at runtime. Additionally, assertThrows can ensure that assertions are thrown at appropriate times given functional arguments to fulfill the requirements. Within AppointmentServiceTest JUnit tests were implemented to ensure that it was true that the supplied data was passed into the Appointment object, that it was false that a duplicate ID could be added to the data structure, and that an error was thrown if an Appointment object was added with a date before the specified limit. Because all these tests were evaluated at runtime, these are dynamic tests.

Categories of testing not implemented within this project were static testing, black-box testing, and experience-based testing. Static testing requires programmers or team members to manually review code in search of errors. This can be an effective method to reduce development time and cost and improve the overall quality of the code. If these modules were completed in a team or pair environment, we could have utilized pair coding and code reviews to check for errors before runtime. As opposed to white-box, black-box testing does not require knowledge of the inner-workings of the software system. Some examples of this type of testing include regression testing, decision table testing, and state transition testing. Finally, experience-based testing is far less rigidly outlined and rely upon the knowledge and experience of developers. If there are not enough adequate specifications to derive test cases or not enough time to design and run formal tests, the user’s and tester’s experience is used to find areas of likely error. Checklist based testing and error guessing are common experience-based techniques in which users or testers seek out error-prone areas within software or systems based on prior knowledge.

My mindset when working on these individual modules was to be patient and focus on understanding the underlying concepts being practiced. I sought other resources online to gain as much insight as possible. I had an overall sense of caution when approaching the first assignment because I had never written JUnit tests before, and it was important to understand the complexity and interrelationships of the code in order to write effective tests. Before writing any code I ensured that I understood the requirements of both the Taks class and the TaskService class. Planning out what my code should do and understanding all requirements before beginning helped prevent me from getting stuck later on or having errors go unnoticed. An example was me incrementally designing the code to create a Task object with the appropriate attributes and methods before ever writing any tests. I ensured that my code could be compiled and executed without error before transitioning into focusing on JUnit tests. Finally, I wrote a single test at a time to ensure each worked properly.

It is natural for bias to be present when reviewing your own work. Because I wrote the code, I have a certain level of attachment to it and will be less likely to desire changing the code as opposed to other individuals reviewing my code. If I were tasked to review my own code, I would not examine it as in-depth as a third party would. Because I wrote the code, I innately assume that it is written in an effective and efficient way. As long as the code is working properly, I am not very likely to seek changes. If another individual were to examine my code, on the other hand, they would have recommendations to make based on their own experience and level of expertise.

Discipline is a key trait to possess for software engineers. It is my responsibility to understand best practices within my field and to always strive to write high-quality code. When writing and testing code, I should never attempt to take shortcuts or to cut corners. Complacence will lead to lower quality code being written, which will produce bugs, longer development time, missed deadlines, and/or higher development costs over time. It seems like a natural occurrence to lose discipline as time goes on and you complete a task more frequently, but I must strive to take accountability of my actions and to never stop focusing on producing code of the highest quality. Errors made due to a lack of effort and discipline will end up costing myself, my employer, and clients time and money. Additionally, I am undergoing a long and arduous task to become a proficient programmer, and I should strive to keep that drive and discipline within me as I progress professionally. I plan to do to by continually learning, seeking input from peers, and not losing the mindset and appreciation of code that I possess currently.